Some Python Projects by Zixuan Li

To access the projects:

https://drive.google.com/drive/folders/10dfcCN-4bi2GI4JtnFD9w18GPlKLtCcA?usp=drive_link

Python Project 1: Four-in-a-Row Game Implementation

In this project, I implemented a Python program that simulates the classic "Four-in-a-Row" game, demonstrating my proficiency in object-oriented programming (OOP), testing, and problem-solving. The game involves two players who take turns dropping pieces into a grid, aiming to get four of their pieces in a row, column, or diagonal to win.

Key Components and Methods

1. Classes and OOP Principles:

- Square: Represents a location in the grid with coordinates and records the occupying player's piece.
- Line: Represents a sequence of squares that can form a row, column, or diagonal.
- Grid: Represents the n-by-n grid of squares and provides functionalities to drop pieces and check for a winning line.
- FourInARow: Manages the game state, including the grid and the turn-taking mechanism.

2. Grid Class Methods:

- is_full: Checks if the grid is filled.
- has_fiar: Determines if there is a line of four consecutive pieces of the same player.
- drop: Allows a piece to be dropped into a specified column and returns the row coordinate where the piece lands.

3. **Testing and Debugging**:

- Doctests and Pytests: Implemented comprehensive tests for each method to ensure correctness and facilitate debugging. For instance, methods like `is_row`,

`is_column`, and `is_diagonal` have specific test cases to validate their functionality.

- **Coverage Check**: Ensured full code coverage using the `check_coverage.py` script, verifying that all lines of code are tested.

4. Helpers and Representation Invariants:

- **Helpers**: Developed helper functions such as `within_grid`, `all_within_grid`, `reflect_vertically`, and `reflect_points` to break down complex tasks into manageable pieces.

- Representation Invariants: Enforced invariants in the `Line` class to maintain consistency and integrity of the game's data structure.

5. Systematic Development:

- Bottom-up Implementation: Built the project from the simplest classes up to more complex ones, ensuring a solid foundation for the overall program.

- IDE Proficiency: Utilized PyCharm features such as hovering for docstrings, navigating to implementations, and auto-testing to enhance development efficiency.

This project not only improves my ability to implement complex logic using Python but also boosts my proficiency in systematic development, testing, and debugging, making me well-equipped for roles that require strong programming skills and attention to detail.

Python Project 2: Grocery Store Simulation

In this project, I developed an event-driven simulation of a grocery store to model customer interactions with different types of checkout lines. The goal of the simulation was to determine the optimal number and types of checkout lines to minimize customer wait times and maximize efficiency.

Key Components and Methods

1. Classes and Object-Oriented Design:

- GroceryStore: Manages the overall simulation, keeping track of all customers and

checkout lines. It initializes the store with a fixed number of regular, express, and self-serve lines based on a configuration file.

- Customer: Represents a customer with a unique name and a list of items to checkout.

- CheckoutLine: Base class for different types of checkout lines. Subclasses include `RegularLine`, `ExpressLine`, and `SelfServeLine`, each with specific rules for customer entry and checkout time calculation.

2. Event-Driven Simulation:

- Events: Defined four types of events—Customer Arrival, Checkout Started, Checkout Completed, and Close Line. Each event has a timestamp indicating when it occurs and specific actions that update the simulation state.

- Event Handling: Events are processed in a loop, with the event having the smallest timestamp being processed first. New events can be generated because of processing an event.

3. Priority Queue for Event Management:

- PriorityQueue Class: Implemented a priority queue to manage the events efficiently. The queue supports inserting events with a given priority and removing the event with the highest priority (earliest timestamp).

- Testing Priority Queue: Developed a robust set of pytest test cases for the `PriorityQueue` class to ensure its correctness and reliability.

4. Simulation Execution:

- Initialization: The simulation starts by reading events from a file and adding them to the priority queue.

- Processing Loop: Continuously processes events from the priority queue, updating the state of the store, customers, and checkout lines, and generating new events as needed.

- Statistics Collection: Throughout the simulation, statistics such as the total number of customers, the timestamp of the final event, and the maximum customer wait time are collected and reported at the end.

5. Helper Functions and Modularity:

- Modular Design: Broke down complex tasks into smaller, manageable functions and methods, promoting code reuse and readability.

- Testing and Debugging: Implemented both doctests and pytests for individual methods and the overall simulation to ensure correctness and facilitate debugging.

This project shows my ability to design and implement complex simulations using Python, leveraging object-oriented principles, event-driven programming, and efficient data structures.

Python Project 3: Blocky Game Implementation

In this project, I developed a Python program for a game called Blocky. The game involves players performing various actions on a board composed of blocks to achieve specific goals. The project demonstrates my ability to work with tree structures, recursion, inheritance, and GUI interactions.

Key Components and Methods

1. Tree Structure and Block Representation:

- Block Class: Represents the fundamental unit of the game board. Each block can be a solid color or subdivided into four smaller blocks.

- Tree Representation: Blocks are organized in a hierarchical tree structure, with the top-level block at level 0 and subdivisions at increasing levels.

2. Rendering and Display:

- _block_to_squares Function: Converts the block structure into a format suitable for rendering on the screen. Utilizes the `renderer.py` module to visually represent the game board.

- Block Rendering: Implemented to display a random-colored block initially and then render changes as the game progresses.

3. Block Actions:

- Rotate: Rotates a block and its children 90 degrees clockwise or counterclockwise.

- Swap: Swaps the left and right or top and bottom children of a block.

- Smash: Subdivides a block into four new randomly colored sub-blocks, potentially recursively.

- Paint: Changes the color of a unit cell to the player's goal color.

- Combine: Turns a block into a leaf based on the majority color of its children.

- Pass: Allows a player to skip their turn without modifying the board.

4. Goals and Scoring:

- BlobGoal and PerimeterGoal Classes: Define two types of goals. BlobGoal aims for the largest connected group of a given color, while PerimeterGoal aims to maximize the color on the board's perimeter.

- Score Calculation: Implemented methods to calculate scores based on the player's goals. Used the `flatten` function to convert the board into a two-dimensional array for easier scoring.

5. Player Types:

- HumanPlayer: Allows a human user to select and perform actions using mouse and keyboard inputs.

- RandomPlayer: A computer player that chooses moves randomly. Implemented using the `generate_move` method to create a random valid move.

- SmartPlayer: A more advanced computer player that generates multiple random moves and selects the one with the highest score. Uses a difficulty level to determine the number of moves to evaluate.

6. Event Handling and Game Loop:

- Process Events: Methods to handle different actions and update the game state accordingly.

- Game Loop: Runs the game, alternating turns between players, and updates the board and scores after each move.

7. Testing and Debugging:

- Pytest and Doctests: Implemented tests to ensure the correctness of methods and the overall game functionality.

- Manual Testing: Played the game to verify that actions and scoring mechanisms worked as expected.

This project enhances my skills in working with complex data structures, implementing recursive algorithms, designing classes with inheritance, and creating interactive

programs with a graphical user interface.